Exception Handling

- Error handling in general
- Java's exception handling mechanism
- The catch-or-specify principle
- Checked and unchecked exceptions
- Exceptions impact/usage
  - Overloaded methods
  - Interfaces
  - Inheritance hierarchies
  - Constructors
Error Handling

- Not all errors can be caught at compile time!

- Help -- run-time error! What next …?

- First ideas:
  - `System.out.println()`
  - `System.err.println()` (much better than the previous)

- Good guess but some errors call for corrective action, not just warning.

- In general, *printing* is a bad idea!

- Better: tell *someone* (not necessarily the user)!
Error Handling, cont.

- Establish return code convention
  - 0 vs. !0 in C/C++
  - `boolean` in Java

- Set value of a global variable
  - Done in many shells.
  - In Java use a public static field in a class.

- Raise an exception, catch it, and act
  - The idea comes from hardware.
  - Modern language support (Java, Python, Lisp, Ada, C++, C#).
General Errors and Error Handling

- Error must be handled where the occur
  - One error in a method can be handled very differently in the clients, this is not a good approach. Repeating handling of the same error.
  - Can be extremely hard to debug.

- To handle an error detailed information on the error must be provided.
  - Where did the error occur (class, method, line number)
  - What type of error
  - A good error message
  - Dump of runtime stack? (too much information?)

- In object-oriented languages errors are represented by objects.
How to Handle Errors

- **Ignore**: False alarm just continue.
- **Report**: Write a message to the screen or to a log.
- **Terminate**: Stop the program execution.
- **Repair**: Make changes and try to recover the error.

To be able to repair would be the best. However, often the best that can be done is the combination of report and terminate.
Java's Exception Handling

• *Exception*: An event that occurs during the execution of a program the disrupts the normal transaction flow.
  - A run-time phenomenon.

• Exception handling is part of the language.
• Exceptions are objects.
• Exceptions are structured in a class hierarchy.
• It is not possible to ignore an exceptions (nice feature?).
  - A method specifies, which exception may occur, the client must anticipate these exceptions, otherwise compile-time error.
• It is sometimes possible to recover to a known good state after an exception was raised.
Java's Exception Handling, cont.

• Java’s object-oriented way to handle errors
  ▪ more powerful, more flexible than using return
  ▪ keywords `try`, `catch`, `throw`, `throws`, `finally`.

• An *exception* is an object that describes an erroneous or unusual situation.

• Exceptions are *thrown* by a program, and may be *caught* and *handled* by another part of the program.

• A program can therefore be separated into a normal execution flow and an *exception execution flow*.

• An *error* is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught.
Motivation for Exception Handling

```java
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            }
        } else {
            errorCode = -2;
        }
    } else {
        errorCode = -3;
    }
    close the file;
    if (theFileDidntClose && errorCode == 0) {
        errorCode = -4;
    } else {
        errorCode = errorCode and -4;
    }
} else {
    errorCode = -5;
}
return errorCode;
```

```java
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

[source: java.sun.com]
Exception Handling Model

- Code where you anticipate a problem:
  - Detect error, probably with an if create a new exception and \texttt{throw} it
  - Alternatively let JVM detect error, create, and throw an exception

  \begin{verbatim}
  public static void main (String args[]) throws exception1, exception2, exception3 {
    . . .
  }
  \end{verbatim}

- Code in client (somewhere in message invocation stack)
  - \texttt{try}, hoping for the best
  - prepare to \texttt{catch} an exception

  \begin{verbatim}
  try{
    // statements that can throws exceptions...
  } catch (exception1) {
    // do stuff
  } catch (exception2) {
    // do stuff
  }
  \end{verbatim}
public class SimpleException extends Exception{}

public class SimpleExample{
    public double calcPrice(int netPrice) throws SimpleException{
        if (netPrice > 100){
            throw new SimpleException(); // to expensive
        }
        return netPrice * 1.25; // add sales tax
    }

    public static void main (String[] args){
        SimpleExample se = new SimpleExample();
        try{
            se.calcPrice(10);
            se.calcPrice(23);
            se.calcPrice(1000);
            se.calcPrice(88); // never called
        }
        catch(SimpleException e){
            System.err.println("Caught SimpleException");
        }
    }
}
Java's Catch or Specify Requirement

- **Catch**
  - A method can catch exception by providing and exception handler.

- **Specify**
  - If a method chooses not to catch, then specify which exceptions are thrown.
  - Exceptions are part of a method's public interface.
Checked/Unchecked Exceptions

- An exception is either *checked* or *unchecked*
  - Checked = checked by the compiler

- A *checked exception* can only be thrown within a try block or within a method that is designated to throw that exception.
  - The compiler will complain if a checked exception is not handled appropriately.

- An *unchecked exception* does not require explicit handling, though it could be processed that way.
  - An an example many run-time exceptions are unchecked exceptions.
Java's Exception Class Hierarchy

Object
   ^
    | Checked
    v
Thrownable
   ^
    | Checked
    v
Error
   |
    v
Subclasses
   |
    v
Unchecked (System and JVM Errors)

Exception
   |
    v
Subclasses
   |
    v
Unchecked

RunTimeException
   |
    v
Subclasses
   |
    v
Unchecked
Java's Exception Class Hierarchy, cont.

- **Throwable**
  - Superclass for all exceptions
  - Two methods for filling in and printing the stack

- **Error**
  - Serious internal errors (should not occur in running programs).
  - Are normally not handled. (report and terminate)
  - Programs should not throw `Error`
  - The catch or specify principle does not apply, because they are so severe.
  - **Examples**
    - Dynamic linking failure
    - Memory shortage
    - Instantiating abstract class
Java's Exception Class Hierarchy, cont.

- **Exception**
  - The base class for most exceptions used in Java programs
  - The catch or specify principle does apply
  - Examples of subclasses
    - `IOException`
    - `ClassNotFoundException`

- **RuntimeException**
  - Not a good name (all exceptions are at run-time)!
  - Commonly seen run-time error
  - The catch or specify principle does not apply, because they are so ubiquitous.
  - Examples
    - Divide by zero/Cast error/Null pointer
The **try** Statement

- To process an exception when it occurs, the line that throws the exception is executed within a *try block*.
- A try block is followed by one or more *catch* clauses, which contain code to process an exception.
- Each catch clause has an associated exception type.

```java
try {
    // statements
}
```
The `catch` Statement

- The catch statement is used for catching exceptions.
- A try statement must be accompanied by a catch statement.
- Try and catch statements can be nested, i.e., try block in try block, etc.

```java
try {
    ...
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Caught first " + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught second " + e.getMessage());
}
```
The **catch** Statement, cont.

- When an exception occurs, processing continues at the first catch clause that matches the exception type.
- The catch statements should be listed in *most-specialized-exception-first* order.

```java
try {
    . . .
} catch (Exception e) { // very general exception
    System.err.println("Caught first " + e.getMessage());
} catch (ArrayIndexOutOfBoundsException e) {
    // will never be called
    System.err.println("Caught second " + e.getMessage());
}
```
The **finally** Clause

- A try statement can have an optional clause designated by the reserved word **finally**.
- If no exception is generated, the statements in the finally clause are executed after the statements in the **try** block complete.
- Also, if an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete.

```java
try {
    // statements that throw exceptions
} catch(<exception>) {
    // do stuff
} finally {
    // code here runs whether or not catch runs
}
```
The **finally** Clause, cont.

```
previous statement

try

FirstException

SecondException

finally

next statement
```
The **finally** Clause, Example

```java
try {
    out = new PrintWriter(new FileWriter("out.txt"));
    // statements that throws exceptions

    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught array error");
    } catch (IOException e) {
        System.err.println("Caught I/O error");
    } finally {
        if (out != null) {
            System.out.println("Closing file");
            out.close();
        }
    }
```
The **throw** Statement

- All methods use the **throw** statement to throw an exception.

```java
public Object pop() throws EmptyStackException {
    Object obj;

    if (size == 0)
        throw new EmptyStackException();

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

[source: java.sun.com]
Exception Propagation

- If it is not appropriate to handle the exception where it occurs, it can be handled at a higher level.
- Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the outermost level.
- A try block that contains a call to a method in which an exception is thrown can be used to catch that exception.
static void method1 throws IOException {
    throw new IOException("Error in method1");
}

static void method2 throws IOException {
    // do stuff, but no catch, just specify
    method1();
}
static void method3 throws IOException {
    // do stuff, but no catch, just specify
    method2();
}

public static void main (String args[]){
    // catch if just specify error to console
    try {
        method3();
    } catch (IOException e){
        // handle the exception from method1
    }
}
Rethrowing an Exception

static void method1 throws IOException {
    throw new IOException("Error in method1");
}
static void method2 throws IOException {
    try {
        method1();
    } catch (IOException e) {
        System.err.println ("Handle partly here");
        throw e; // 1st method
        // throw e.fillInStackTrace; // 2nd method
        // throw new IOException ("new one"); // 3th method
    }
}
public static void main (String args[]){
    // catch if just specify error to console
    try {
        method2();
    } catch (IOException e) {
        System.err.println ("Handle rest here");
    }
}
Creating New Exceptions

- Requires careful design (part of the public interface).
- Can an existing Exception be used?
- Choose the correct superclass.
- Choosing the name
  - The most important thing for new exceptions.
  - Tends to be long and descriptive (ArrayIndexOutOfBoundsException)
- Code for exception class typically minimal

- Naming convention:
  - All classes that inherit from Exception has 'Exception' postfixed to their name.
  - All classes that inherit from Error has 'Error' postfixed to their name.
Creating New Exceptions, Example

class SimplestException extends Exception {
    // empty method body okay, just give it a good name
}

class SimpleException extends Exception {
    SimpleException () { super(); } // default constructor
    SimpleException (String str) { super(str); }
}

class ExtendedException extends Exception {
    private static int counter = 0;  // no of exceptions
    ExtendedException () { super(); counter++; }
    ExtendedException (String str) {
        super(str); counter++; }
    ExtendedException (String str, int no) {
        super(str);
        instanceNo = no;
        counter++; }
}
Overloading and Exception

- Methods cannot be overloaded based on exception specification.

```java
public class OverloadedMethod{
    /** An overloaded method */
    public int calc(int x) throws SimpleException {
        return x;
    }
    /** NOT allowed */
    public int calc(int y) throws AnotherException {
        return y;
    }
    /** Is allowed */
    public int calc(int x, int y){
        return x + y;
    }
    public static void main(String[] args){
        OverloadedMethod om = new OverloadedMethod();
        System.out.println(om.calc(3));
    }
}
```
Interfaces and Exception

• Exceptions can naturally be specified for methods in interfaces

```java
public interface InterfaceException{
    int calc(int x) throws SimpleException;
    // not allowed
    //int calc(int y) throws AnotherException;
    int calc(int x, int y)
        throws SimpleException, AnotherException;
}
```
Inheritance and Exceptions

- If base-class method throws an exception, derived-class method may throw that exception or one derived from it.

- Derived-class method cannot throw an exception that is not a type/subtype of an exception thrown by the base-class method.
  - Otherwise subclass cannot be upcasted to base-class.

```java
class BaseException extends Exception{}
class DerivedException extends BaseException{}

class A { void f() throws BaseException{} }
class B extends A { void f() throws DerivedException{} }

// not allowed compile-error
class C extends B { void f() throws AnotherException{} } }
```
Inheritance and Constructors

• Constructors can throw exceptions
• Subclass constructor cannot catch exception throws by base class constructor.

```java
class A{
    int i;
    A(int j) throws SimpleException{
        if (j < 0){ throw new SimpleException(); }
        i = j;
    }
}
class B extends A {
    B(int j) throws SimpleException, AnotherException{
        // cannot have try block here
        super(j);
        if (j > 100){ throw new AnotherException(); } } }
```
Guidelines

• Do not use exceptions for normal control flow!
  ▪ Slows down the program
• Do use exceptions to indicate abnormal conditions!
• Handle the error (fully or partially) if you have enough information in the current context. Otherwise, propagate!
• Handle group of statements
  ▪ Do not encompass every single statement in a try block
• Use exceptions in constructors!
• Do something with the exceptions your code catches!
• Clean up using finally.
Summary

- The manner in which an exception is processed is an important design consideration.

- Advantages of Exceptions
  - Separates error handling from “regular” code.
  - Propagation of errors up the call stack.
    - Handle error in a context
  - Grouping of error type and differentiation of errors.
    - Overview
    - Reuse of error handling code